User-defined routines

DB2 Information Management Software

http://www-136.ibm.com/developerworks/db2

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Before you start 2
2. Using user-defined functions 5
3. Working with external functions 17
4. Working with stored procedures 25
5. Using the DB2 Development Center 33
6. Advanced use of the Development Center 44
7. Conclusion 49

Section 1. Before you start

What is this tutorial about?

In this tutorial, you'll learn about using user-defined routines with the DB2 Universal Database. You'll see how you can utilize programming logic within DB2 to simplify and speed up your applications. We'll discuss the following topics:

- User-defined functions and how to use them
- Stored procedures
- The DB2 Development Center tool

This is the seventh in a series of seven tutorials that you can use to help prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The material in this tutorial primarily covers the objectives in Section 7 of the exam, entitled "User-defined routines." You can view these objectives at: *http://www.ibm.com/certify/tests/obj703.shtml*.

You do not need a copy of DB2 Universal Database to complete this tutorial. However, you can download a free trial version of *IBM DB2 Universal Database* Enterprise Edition for reference.

Who should take this tutorial?

To take the DB2 UDB V8.1 Family Application Development exam, you must have already passed the DB2 UDB V8.1 Family Fundamentals exam (Exam 700). You can use the DB2 Family Fundamentals tutorial series (see Resources on page 49) to prepare for that test. It is a very popular tutorial series that has helped many people understand the fundamentals of the DB2 family of products.

This tutorial is one of the tools that can help you prepare for Exam 703. You should also review Resources on page 49 at the end of this tutorial for more information.

This tutorial is designed to help you study for the user-defined routines section of the DB2 Family Application Development exam. For this tutorial to be useful, you should have a solid understanding of how SQL is used and how it works with a database. An understanding of programming methodologies and flow of control usage would also help. You should also have a base knowledge of:

- Databases
- Database objects
- Application compilation

Terminology review

Before you begin this tutorial, you need to be familiar with the concept of an *SQL access plan*. This is a set of steps the DB2 engine uses to execute an SQL statement. It includes the indexes that are used, the times at which fields are retrieved from the database tables, and the order in which the steps of the query are executed. The access plan is created by a database engine based on an SQL statement sent to that engine.

We'll be using the term throughout this tutorial, so it's important to make sure you understand it up front.

About the author

Drew Bradstock is an engagement manager for IBM Competitive Migration Services. He was previously part of the IBM Data Management Business Partner Enablement team, where he worked with IBM's partners to migrate their applications to DB2. As part of this team, he was a coauthor of the DB2 textbook *DB2 SQL Procedural Language for Linux, UNIX, and Windows.* He was also a coauthor of the IBM Redbook "Scaling DB2 UDB on Windows Server 2003." Drew has specialized in performance-tuning databases and migrating application code and architectures.

Drew is an IBM-certified DB2 V8.1 Advanced Administrator, Business Intelligence expert and Application Developer. He was also a member of the team that developed the DB2 V8 for Linux, UNIX, and Windows (LUW) certification exams.

You can contact Drew at *drewkb@ca.ibm.com*.

Notices and trademarks

Section 2. Using user-defined functions

What are user-defined functions?

User-defined functions, or UDFs as they are commonly known, are functions defined by the user aaqqthat run in the database engine. The functions are used to simplify the logic within SQL statements. Instead of repeatedly including common or complex logic in multiple statements, the developer can isolate that logic in a single function.

Traditionally, people think of a function as a program that takes in some inputs and then returns a result. Within a database, you have a great deal more flexibility and possibilities in what you can do with a function. There are a number of different types of functions, and also different languages you can write them in.

Here are the types of available functions:

- **SQL Scalar**: Returns a single data type value using SQL
- **SQL Row**: Returns a row of data using SQL
- **SQL Table**: Returns an SQL table based on SQL
- **External Scalar**: Returns a single data type using an external language
- **Sourced** or **Template**: Are based on another function or template

And here are the programming languages that you can write them in:

- SQL (for functions written using one or more SQL statements)
- C/C++
- The Java language
- SQLJ
- Visual Basic

Function types

There are two distinct types of functions. The more common type is an *internal function*, which is a function written using SQL statements. Functions can also be written as references to external components of code, be they in the Java language, C, or other languages. These functions are referred to as *external functions*. The code for internal functions is stored in the database catalog tables, while external functions are just references to locations where the compiled code is stored.

One important difference between the two types of functions is the location at which you compile them. An SQL function can be compiled either remotely or

locally, since it will be stored in the database server's catalog tables. An external function, however, is not as flexible. It must be compiled on the server, since the location of the compiled code is then stored by the database server. If you compile it on a client, then the code is stored on the client. The local client would be able to execute the function, since the DB2 client would be able to find the program. However, the server or any other clients would return an error, since the compiled code would not be accessible to them. You could compile the function on the client and copy it to the server at the exact same location to avoid this problem. But for that to work, the operating system and underlying hardware need to be the same on both systems, since the code would have been compiled by a specific compiler for the client system and may not work with another computer/OS combination.

Creating a function

Internal functions are easy to create and use in your applications. A simplified version of the syntax diagram is displayed below and is followed by an explanation of its major parts.

```
>>-CREATE FUNCTION--function-name------------------------------>

>--(--+---------------------------------+--)--*----------------->
      | .-,---------------------------. |
      | V                             | |
      '---parameter-name--data-type1-+-'

>--RETURNS--+-data-type2-----------------+--*------------------->
            '-+-ROW---+--| column-list |-'
              '-TABLE-'

                                    .-LANGUAGE SQL-.
>--+-------------------------+--*--+--------------+--*---------->
   '-SPECIFIC--specific-name-'

   .-NOT DETERMINISTIC-.     .-EXTERNAL ACTION----.
>--+-------------------+--*--+--------------------+--*---------->
   '-DETERMINISTIC-----'     '-NO EXTERNAL ACTION-'

   .-READS SQL DATA---------.
>--+------------------------+--*--+-----------------+--*-------->
   +-CONTAINS SQL-----------+
   |                        |
   '-MODIFIES SQL DATA------'

>--| SQL-function-body |------------------------------------->

column-list:

      .-,-----------------------.
      V                         |
|--(----column-name--data-type3-+--)---------------------------|

SQL-function-body:
```

```
|--+-RETURN Statement-----------+------------------------------|
   '-dynamic-compound-statement-'
```

The major components of an SQL function are listed below. We'll look at some examples on the next panel.

- **Function name**: Name of the function.
- **RETURNS *type***: The type of function that is called. The types available are *scalar*, *row*, or *table*. We'll look at each type in detail in Scalar functions on page 12,Row functions on page 14, andTable functions on page 15. The scalar value is actually a data type, not the word "SCALAR".
- **Specific**: You can assign the compiled function a specific name rather than having DB2 assign a system-generated unique name for it.
- **Deterministic**: This specifies whether the function will return the same result each time it is run. Deterministic functions include math functions and functions that are not dependent on data in tables or changing data sources.
- **External Action**: Specifies if the function has any impact on external programs.
- **Has SQL**: Specifies how the function interacts with the database.
- **Called on Null Input**: Specifies if the function should be called if any of the input values are null.
- **SQL function body**: This is the meat of the function, where the logic is contained.

CREATE FUNCTION sections

This panel shows a number of examples to help you better understand what each of the terms in the `CREATE FUNCTION` command mean.

RETURNS

The `RETURNS` specification controls the type of function that will be created. The three major types are scalar, row, and table. A scalar function, like the following example, only returns a single data type value:

```
CREATE FUNCTION TAN (X DOUBLE)
    RETURNS DOUBLE
    LANGUAGE SQL
    CONTAINS SQL
    NO EXTERNAL ACTION
    DETERMINISTIC
    RETURN SIN(X)/COS(X)
```

For more on scalar functions, see Scalar functions on page12 .

A row function, like the following example, separates a user-defined type into its

different components:

```
CREATE FUNCTION FROMPERSON (P PERSON)
    RETURNS ROW (NAME VARCHAR(10), FIRSTNAME VARCHAR(10))
    LANGUAGE SQL
    CONTAINS SQL
    NO EXTERNAL ACTION
    DETERMINISTIC
    RETURN VALUES (P..NAME, P..FIRSTNAME)
```

For more on row functions, see Row functions on page14 .

A table function, like the following example, will return zero or more rows of a table. The table can be created from SQL or from application logic.

```
CREATE FUNCTION DEPTEMPLOYEES (DEPTNO CHAR(3))
    RETURNS TABLE (
            EMPNO CHAR(6),
        LASTNAME VARCHAR(15),
        FIRSTNAME VARCHAR(12))
    LANGUAGE SQL
    READS SQL DATA
    NO EXTERNAL ACTION
    DETERMINISTIC
    RETURN
        SELECT EMPNO, LASTNAME, FIRSTNME
        FROM EMPLOYEE
        WHERE EMPLOYEE.WORKDEPT = DEPTEMPLOYEES.DEPTNO
```

For more on table functions, see Table functions on page15 .

SPECIFIC

The `SPECIFIC` command is used to ensure that you have a specific name for the function that allows you to remember what the compiled function is. This specific name can then be referenced when you are sourcing, dropping, or commenting the function. This is useful when you have overloaded the function and have more than one version of the function with different data inputs.

The following two examples illustrate such overloading. The first function joins two integer values by adding them together. The second joins the string `new_` to the input string.

```
CREATE FUNCTION joinData (x INT, y INT)
    RETURNS DOUBLE
        SPECIFIC join_int2
    LANGUAGE SQL
    CONTAINS SQL
    NO EXTERNAL ACTION
    DETERMINISTIC
    RETURN x + y;
*******
CREATE FUNCTION joinData (x VARCHAR(10)
```

```
    RETURNS VARCHAR(15)
        SPECIFIC join_str
    LANGUAGE SQL
    CONTAINS SQL
    NO EXTERNAL ACTION
    DETERMINISTIC
    RETURN 'new_' || x;
```

DETERMINISTIC

The `DETERMINISTIC` option is used to specify whether the function always returns the same value. This information can then be used by DB2 to optimize how the function is called, since DB2 would already know the value of the function if it had been run once before and is deterministic. If a function uses any special registers, or it calls any nondeterministic functions, it must be nondeterministic.

Here's an example of a deterministic function:

```
CREATE FUNCTION joinData (x INT, y INT)
    RETURNS DOUBLE
    LANGUAGE SQL
    CONTAINS SQL
    NO EXTERNAL ACTION
    DETERMINISTIC
    RETURN x + y;
```

And here's an example of a nondeterministic function:

```
CREATE FUNCTION futureDate (x INT)
     RETURNS DATE
     LANGUAGE SQL
     CONTAINS SQL
     NO EXTERNAL ACTION
     NOT DETERMINISTIC
     RETURN CURRENT DATE + x MONTHS;
```

EXTERNAL ACTION

This option specifies whether the function will change any objects outside the database system. This option must be set to `EXTERNAL ACTION` if the function calls any functions that have an external impact. An example would be a function that modifies a flat file or alters data in an external source, such as a genome data file.

HAS SQL

This option lets DB2 know how the function is going to interact with the database. You have a number of options here:

° `CONTAINS SQL`: Indicates that SQL statements that neither read nor modify SQL data can be executed by the function.

- `READS SQL DATA`: Indicates that SQL statements that do not modify SQL data can be executed by the function.
- `MODIFIES SQL DATA`: Indicates that all SQL statements supported in the dynamic-compound statement can be executed by the function. This option was added in DB2 V8.1 Fixpak 4. Previously, SQL functions could not modify the data in the database.

Function rules

There are a number of restrictions on functions that you should be aware of when dealing with DB2 SQL functions:

- If an SQL function contains multiple references to any of the date or time special registers, all references return the same value. This will be the same value returned by the register invocation in the statement that called the function.

- The body of an SQL function cannot contain a recursive call to itself or to another function or method that calls it, since such a function could not exist to be called.

Privileges

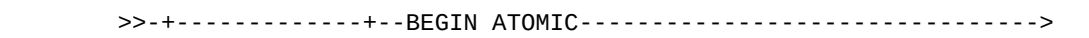
When a function is created, the `EXECUTE` privilege is automatically given to the definer of the function. The definer is also given the right to drop the function, even if it does not have `DBADM` authority. Along with these authorities, the definer of the function is given the `WITH GRANT OPTION`, which allows it to pass on the authorities it has on the function to others. The function definer must have the required privileges explicitly and cannot have them as result of being part of a security `GROUP`.

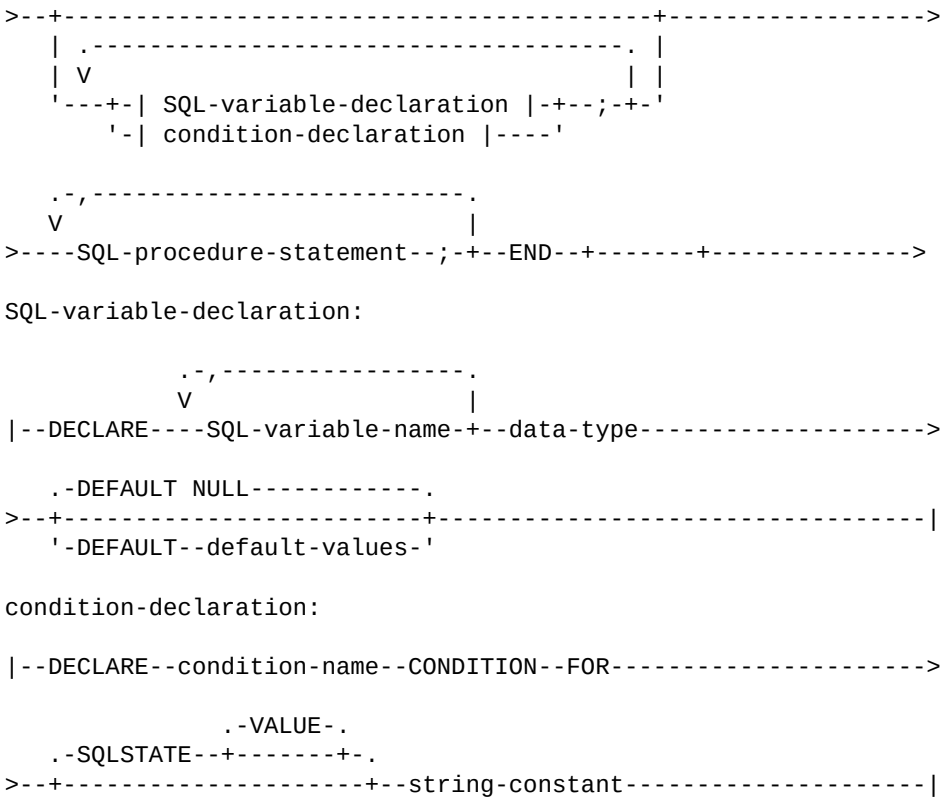
Using compound statements in functions

What is a compound statement?

You are no longer limited to a single line of code for your SQL function logic. A compound SQL statement is a group of statements contained within a `BEGIN-END` block. The SQL statements are then treated as one atomic unit. This block of statements can be dynamically prepared.

Here's the syntax diagram for a compound SQL block:





The major parts of the compound statement are:

DECLARE: This allows you to declare variables within an SQL block. The data type can be any user-defined type or standard SQL data type. If a default value is not given for a data type, it is automatically set to null when it is declared. Here are some examples:

```
DECLARE drewInt INTEGER;
DECLARE drewChar CHAR(6);
DECLARE drewI2 INTEGER DEFAULT 0;
DECLARE booBoo VARCHAR(100) DEFAULT NULL;
```

CONDITIONAL HANDLING: This option is currently not available for functions.

SQL control statements: The following statements can be used within a compound statement:

- FOR
- GET DIAGNOSTICS
- IF
- ITERATE
- LEAVE
- SIGNAL
- WHILE

More information can be found on these statements in the DB2 online help.

The following SQL statements can be issued:

- fullselect (a common-table-expression can precede the fullselect)
- Searched `UPDATE`
- Searched `DELETE`
- `INSERT`
- `SET` variable statement

Working with stored procedures on page 25 covers the different parts of SQL compound statements and provides some more detailed examples. There are a number of differences between the use of procedural SQL in functions and in procedures.

Scalar functions

The different function types offer you different ways to return data to your SQL statements. It is important to understand how the function types can be used, and also how easily complex functions can be written.

An *SQL scalar* function is a traditional function that is probably the first type you think of when you hear the word function. It returns a single value in any of the data types within DB2. This function is normally constructed using an SQL statement.

The simple example below illustrates how you can hide the logic of your programs in a function. The function `changeSal` is created with a single line of real code: `RETURN sal * 2`. The remainder of the lines are the function definition. The function takes in as input the salary figure of an employee, or any other double field. It also works with other number values, since DB2 will implicitly cast them for you.

Here's how you'd create your function:

```
CREATE FUNCTION changeSal (v_sal double)
     RETURNS DOUBLE
     LANGUAGE SQL
     CONTAINS SQL
     NO EXTERNAL ACTION
     DETERMINISTIC
     RETURN v_sal * 2;
```

And here's how you'd use it:

```
SELECT empno,
        changeSal(salary) as newSalary
FROM employee
WHERE edlevel > 12;
```

```
Result from the DB2 sample database:
EMPNO  NEWSAL
------ ----------------------
000030               87975.00
```

Scalar functions are normally more complicated than this, though, and can involve full SQL statements. Let's create a more complex function:

```
CREATE FUNCTION edCount (v_edLevel double)
     RETURNS INT
     LANGUAGE SQL
     READS SQL DATA
     NO EXTERNAL ACTION
     RETURN SELECT count(*)
                FROM employee
                WHERE edLevel = v_edLevel;
```

Now, let's use it in an SQL statement:

```
SELECT edLevel,
        edCount(edLevel)
FROM employee
GROUP BY edlevel;
```

DB2 takes the SQL function and *in-line* it into the SQL statement in which the function is called. When you in-line a function, the function call in the calling SQL statement is actually replaced by the function text. As a result, the DB2 optimizer can then create an optimized access plan for the entire statement and not just a section of it. This leads to better overall access plans.

Let's look again at the first sample SQL statement:

```
SELECT empno,
        changeSal(salary) as newSalary
FROM employee
WHERE edlevel > 12;
```

DB2 rewrites it so it looks like this:

```
SELECT empno,
        sal * 2 as newSalary
FROM employee
WHERE edlevel > 12;
```

The second example's in-lining is much more complicated despite how simple the initial SQL statement appears. This example clearly illustrates the benefit of moving logic into the function instead of your own SQL statements. Here's the original SQL statement again:

```
SELECT edLevel,
        edCount(edLevel)
```

```
FROM employee
GROUP BY edLevel;
```

And here's what it looks like after DB2 rewrites it:

```
SELECT Q3.$C0 AS "EDLEVEL", Q6.$C0
FROM
        (SELECT Q2.$C0
        FROM
                (SELECT Q1.EDLEVEL
                FROM DB2ADMIN.EMPLOYEE AS Q1) AS Q2
        GROUP BY Q2.$C0) AS Q3,
        (SELECT COUNT(* )
        FROM
                (SELECT $RID$
                FROM DB2ADMIN.EMPLOYEE AS Q4
                WHERE (Q4.EDLEVEL = DOUBLE(Q3.$C0))) AS Q5)AS Q6
```

Row functions

With functions, you can also return an entire row of data, instead of just a single data type. The row function does not return a row of data, as you might expect from its name. It is instead used to transform structured data types into their individual components. User-defined types, or UDTs, are data types defined by the user that contain references to one or more DB2 data types. (UDTs are covered in detail in the sixth tutorial in this series, see Resources on page 49.) You would therefore only use row functions if you have UDTs in your database. Row functions can only be defined as SQL functions, and you cannot use the other programming languages that are normally allowed for DB2 functions.

The following example illustrates how you can use a row function to determine the last name and first name of a person. The `person` object in the example is a UDT that contains both the `lastName` and `firstName` fields. Our function can be used to break apart the information you need from the `person` type.

Here's the command creating the `person` type:

```
CREATE TYPE person AS (
        lastname varchar(15),
        firstname varchar(12))
        MODE DB2SQL
);
```

Next, create a table with the `person` information:

```
CREATE TABLE empInfo (
        personInfo person,
        empno char(6)
);
```

Then create the row function:

```
CREATE FUNCTION FROMPERSON (P PERSON)
  RETURNS ROW (LNAME VARCHAR(15), FNAME VARCHAR(12))
  LANGUAGE SQL
  CONTAINS SQL
  NO EXTERNAL ACTION
  DETERMINISTIC
  RETURN VALUES (P..LASTNAME, P..FIRSTNAME)
```

Table functions

One of the most powerful aspects of DB2 functions is their ability to return an entire table of data instead of a single value. This opens up many sources of information that you can use in your SQL statements. Instead of referring to a database table, you could instead write a C function that referred to a live data stream such as stock market data.

Table functions are actually quite easy to write and to reference. Instead of referencing a single data value, as in scalar functions, you reference multiple variables that refer to the different columns that will be returned in the table data, as in the following example:

```
CREATE FUNCTION DEPTEMPLOYEES (DEPTNO CHAR(3))
    RETURNS TABLE (EMPNO CHAR(6),
                   LASTNAME VARCHAR(15),
                   FIRSTNAME VARCHAR(12))
    LANGUAGE SQL
    READS SQL DATA
    NO EXTERNAL ACTION
    RETURN
       SELECT EMPNO, LASTNAME, FIRSTNME
         FROM EMPLOYEE
         WHERE EMPLOYEE.WORKDEPT = DEPTEMPLOYEES.DEPTNO
```

There are restrictions to this: You must know the data types and number of parameters that your SQL table will return. If you have a function that refers to the entire table's columns and you add an additional column, then your function may not work properly. For instance, suppose we create the following table:

```
CREATE TABLE testTab (
        varOne int,
        varTwo int
);
```

This function will work properly with that table:

```
CREATE FUNCTION returnAllTest (v_v1 int)
        RETURNS TABLE (v_varOne INT,
                                        v_varTwo INT)
        LANGUAGE SQL
        READS SQL DATA
        NO EXTERNAL ACTION
        RETURN
                SELECT *
                FROM testTab
                WHERE varOne = v_v1;
```

But now we add a new column to the table:

```
ALTER TABLE testTab ADD varThree int;
```

Our function call will no longer return the expected value of all three columns, but will instead return just the columns that were defined when it was created. This is because the `'*'` used in the function definition is expanded out at creation time and not at execution. Here's the SQL output for the modified table:

```
select * from testtab

VARONE      VARTWO      VARTHREE
----------- ----------- -----------

  0 record(s) selected.
```

And here's the function output:

```
select * from table ( returnAllTest(1)) as test

V_VARONE    V_VARTWO
----------- -----------

  0 record(s) selected.
```

Section 3. Working with external functions

Using external functions

The ability to reference external programs as functions gives you more capabilities in your SQL. By building functions in C or the Java language, you can perform very complex string manipulation that would have required multiple (and confusing) SQL statements to duplicate. You can also process outside data sources or perform external actions outside of the database with these programming languages. For a list of supported languages, see What are user-defined functions? on page 5.

In order to be able to create an external function, you must have at least one of the following privileges:

- `SYSADM` or `DBADM` authority
- `CREATE_EXTERNAL_ROUTINE` authority on the database and at least one of:
  - `IMPLICIT_SCHEMA` authority on the database, if the schema name of the function does not refer to an existing schema.
  - `CREATEIN` privilege on the schema, if the schema name of the function refers to an existing schema.

The functions can run either *unfenced* -- that is, inside the database engine -- or *fenced* -- that is, in memory outside of the database engine. The advantage to having an unfenced function is that it shares the memory area with the engine and can communicate it with it more quickly. However, if an unfenced function has not been written properly, a memory leak may occur. If this happens, the memory leak could start overwriting the memory associated with the DB2 engine, which could have horrible repercussions. If you are writing C or C++ functions it is highly recommended that you write them as fenced.

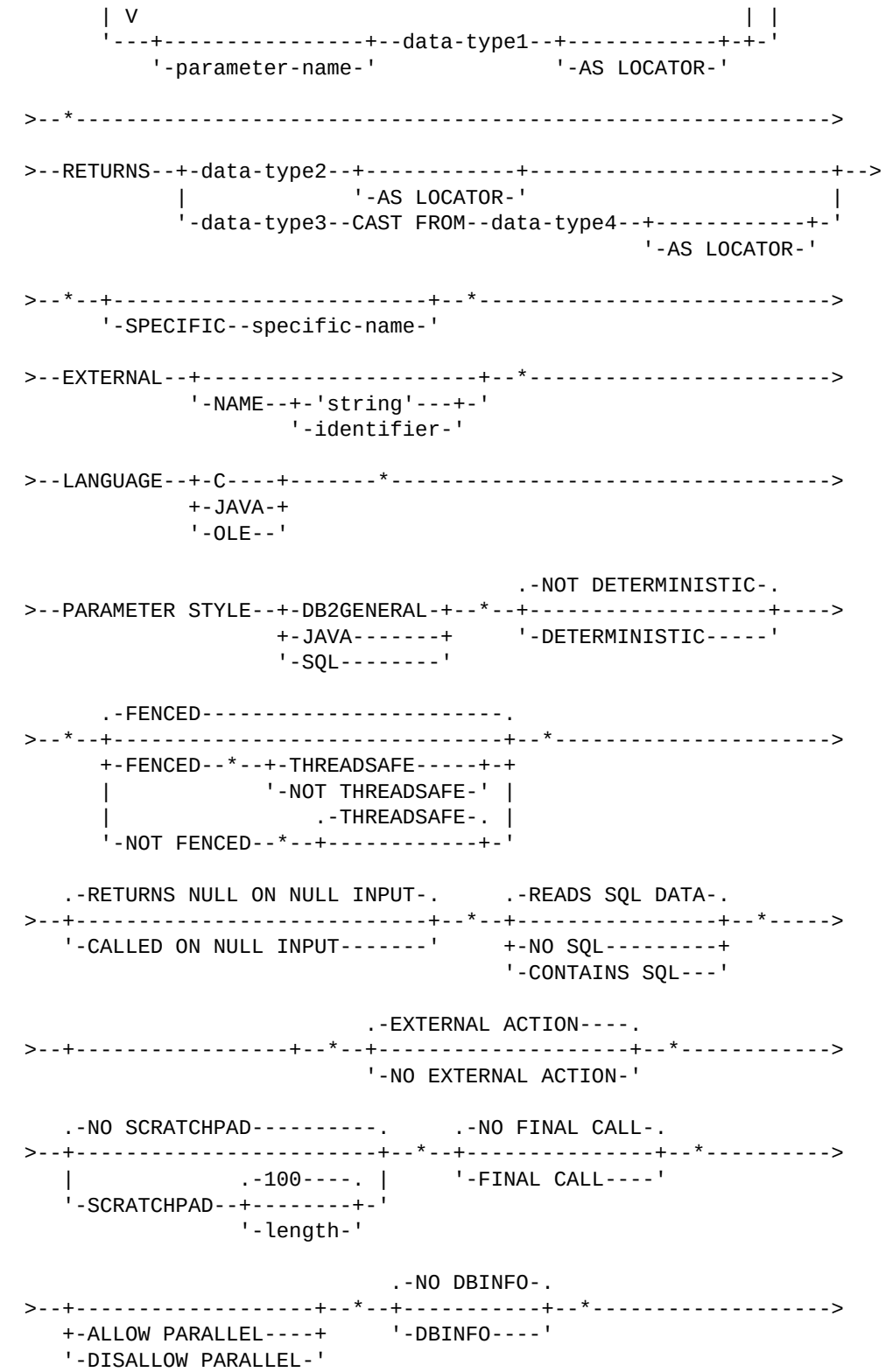
To create an unfenced procedure, you will need one of the following privileges:

- `CREATE_NOT_FENCED_ROUTINE` authority on the database
- `SYSADM` or `DBADM` authority

Creating external functions

Here's a simplified syntax diagram for creating an external function:

```
>>-CREATE FUNCTION--function-name------------------------------>

>--(--+-------------------------------------------------+--)-->
      | .-,---------------------------------------. |
```



We explored the majority of the components of the `FUNCTION` command in Creating a function on page 6. There are a few new components that are particular to external functions, and we'll discuss them here.

- **Language**: This identifies the language that the function is written in.
- **Parameter style**: This clause is used to specify the conventions used for

passing parameters to and returning the value from functions.

- **External action**: This determines if external actions can be performed by the function.
- **Scratchpad**: The scratchpad is used as a memory area where data can be stored by the function.
- **Final call**: This is used to deallocate memory that is being utilized by the function.

Major external function options

One of the most important mandatory parameters in an external function is the `EXTERNAL NAME` option. This controls where DB2 will look for the compiled function code.

C functions

In C, the portion of the command is:

```
>>-'--+-library_id-------+--+------------+--'------------------>|
      '-absolute_path_id-'  '-!--func_id-'
```

The different components of it are as follows:

- **library_id**: This is the location of the library containing the function. The database manager will look for the library in the `.../sqllib/function` directory (on UNIX-based systems) or the `...\sqllib\instance_name\function` directory (on Windows operating systems), as specified by the `DB2INSTPROF` registry variable. The library_id will then be appended onto the function directory location. For example, if `drewFunc` were the library ID, then the database manager would look for the function in library `/u/production/sqllib/function/drewFunc`, assuming the instance was installed in `/u/production`.

- **absolute_path_id**: Identifies the full path name of the file containing the function.

- **! func_id**: Identifies the entry point name of the function to be invoked. The `!` serves as a delimiter between the library ID and the function ID. In a UNIX-based system, for example, `mymod!funcDrew` directs the database manager to look for the library `$inst_home_dir/sqllib/function/mymod` and to use entry point `funcDrew()` within that library. Under Windows operating systems, `mymod!funcDrew` directs the database manager to load the `mymod.dll` file and call the `funcDrew()` function in the dynamic link library (DLL).

Java functions

The syntax in the Java language for this command is:

```
>>-'--+----------+--class_id--+-.-+--method_id--'-------------->|
      '-jar_id :-'            '-!-'
```

The different components are:

° **jar_id**: Identifies the JAR identifier given to the JAR collection when it was installed in the database. It can be either a simple identifier, such as `myJar`, or a schema-qualified identifier, such as `mySchema.myJar`.

° **class_id**: Identifies the class identifier of the Java object. If the class is part of a package, the class identifier part must include the complete package prefix. For example, if you use `myPacks.UserFuncs`, the Java Virtual Machine will look in the directory `.../myPacks/UserFuncs/` (or `...\myPacks\UserFuncs\` on Windows) for the classes.

° **method_id**: Identifies the method name of the Java object to be invoked.

OLE functions

The OLE syntax for this command is:

```
>>-'--+-progid-+--!--method_id--'----------------------------->|
      '-clsid--'
```

The different components are:

° **progid**: Identifies the programmatic identifier of the OLE object.

° **clsid**: Identifies the class identifier of the OLE object to create. It can be used as an alternative for specifying a `progid` in case an OLE object is not registered with a `progid`. The `clsid` has the form `{nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}`, where n is an alphanumeric character.

° **method_id**: Identifies the method name of the OLE object to be invoked.

Function definition examples

Let's start with an example in C. The example below calls as input a single `SMALLINT` parameter and then returns as output a `SMALLINT` field as well. No

SQL is called during the execution of the function.

```
CREATE FUNCTION ntest1 (SMALLINT)
    RETURNS SMALLINT
    EXTERNAL NAME 'ntest1!nudft1'
    LANGUAGE C    PARAMETER STYLE SQL
    DETERMINISTIC   NOT FENCED   NULL CALL
    NO SQL   NO EXTERNAL ACTION
```

Now let's try a Java language example. The following example calls the `findc` function. No SQL is called in the function.

```
CREATE FUNCTION findc ( CLOB(100K))
    RETURNS INTEGER
    FENCED
    LANGUAGE JAVA
    PARAMETER STYLE JAVA
    EXTERNAL NAME 'javaUDFs.findCnst'
    NO EXTERNAL ACTION
    CALLED ON NULL INPUT
    DETERMINISTIC
    NO SQL
```

Binding a function

If you create an external function that executes SQL, you will have to PREPARE and `BIND` the application that will be connected to the database.

The PREPARE command checks that the embedded static SQL in your application is properly formed. For more details on this command, check the DB2 online help under "`PRECOMPILE` Command."

The `BIND` command invokes the bind utility, which prepares SQL statements stored in the bind file generated by the precompiler and creates a package that is stored in the database. Further information on this step can be found in the DB2 online help under "`BIND` Command."

Let's look at an example. The following C program retrieves the application ID for the application that calls it. The program simply retrieves the `appl_id` field from the `dbinfo` data structure. This information cannot be extracted using SQL, so the function can be useful. Here's the C code

```
#include <string.h>
#include <sqludf.h>

void SQL_API_FN getApplicationId(
        SQLUDF_CHAR *applId,  SQLUDF_NULLIND *applId_ind,
        SQLUDF_TRAIL_ARGS,  SQLUDF_DBINFO *dbinfo)
```

```
{
    strncpy(applId, dbinfo->appl_id, 128);
    *applId_ind = 0;
}
```

To transform this function from C code to a state where it can be called in SQL, you need to walk through the following steps:

1. Connect to the database
2. PREP the .sqc function file
3. BIND the .bnd function file
4. Compile the .c function file
5. Create the external function in DB2

The next panel covers these steps in detail.

Binding a function: Example

This example script will prepare and bind our sample function from the previous panel. The script will run on a Windows system, but a C compiler must first be installed and configured. (See Creating a stored procedure on page 26 and Resources on page 49 for more on C compilers.)

```
@ echo off
rem -----------------------------------------
rem Script to create the C function
rem  crFunc.bat databaseName [user [pwd]]
rem -----------------------------------------

rem Default compiler is set to Microsoft Visual C++
rem Microsoft C/C++ Compiler
set BLDCOMP=cl
set PROG=application_Id

rem call sdkvars
rem Precompile and bind the program.
rem Connect to a database.
if "%1" == "" goto error
if "%2" == "" goto case1
if "%3" == "" goto case2
if "%4" == "" goto case3
goto error

:case1
db2 connect to %1
if NOT %ERRORLEVEL% == 0 GOTO connectError
goto prep

:case2
db2 connect to %1 user %2
if NOT %ERRORLEVEL% == 0 GOTO connectError
```

```
goto prep

:case3
db2 connect to %1 user %2 using %3
if NOT %ERRORLEVEL% == 0 GOTO connectError
goto prep

:prep
db2 prep %PROG%.sqc bindfile

:bind
rem bind step
db2 bind %PROG%.bnd
if NOT %ERRORLEVEL% == 0 GOTO errorBind
db2 connect reset

rem Compile the program.
%BLDCOMP% -Zi -Od -c -W2 -DWIN32 -MD %PROG%.c
if NOT %ERRORLEVEL% == 0 GOTO errorCompile

rem Link the program.
link -debug -out:%PROG%.dll -dll %PROG%.obj db2api.lib -def:%PROG%.def
if NOT %ERRORLEVEL% == 0 GOTO errorLink

rem Copy the routine DLL to the 'function' directory
copy %PROG%.dll "%DB2PATH%\function"
if NOT %ERRORLEVEL% == 0 GOTO errorCopy
goto exit

:connectError
echo The connection to the database failed.  Please check the database,
echo username and password.
goto :error

:errorBind
db2 connect reset
echo The application failed to prep and bind the program to the database.
goto :error

:errorCompile
echo The program failed to compile.  Please check that the sdkvars program has
echo been run.
goto :error

:errorLink
echo The program failed to link the files together.
goto :error

:errorCopy
echo The dll file could not be copied to the %DB2PATH%\FUNCTION directory.
goto :error

:error
echo Usage crFunc.bat databasename [user [pwd]]

:exit
@echo on
```

Once you've run this script, the function can be created in the database:

```
CREATE FUNCTION application_id()
   RETURNS VARCHAR(128)
   SPECIFIC applId  EXTERNAL NAME 'application_id!getApplicationId'
   NOT FENCED  LANGUAGE C  PARAMETER STYLE SQL  DETERMINISTIC
   NO SQL  NO EXTERNAL ACTION  ALLOW PARALLEL  DBINFO
```

The function can now be called in any SQL statement, like so:

```
VALUES(application_id())
```

Section 4. Working with stored procedures

What are stored procedures?

Originally, stored procedures were complex C programs used to process and execute multiple SQL commands on the database server. The creation of these programs was quite complex and required in-depth knowledge of C/C++ programming and database interface languages such as the DB2 Call Level Interface (CLI).

As databases evolved, the stored procedures languages grew into procedural languages using SQL. DB2 uses a language called SQL PL, Oracle uses PL/SQL, and SQL Server and Sybase use Transact-SQL (or T-SQL for short). These languages are quite different, but they all follow a common framework of keywords, control statements and behaviour. There is a section in the SQL standard that specifies the accepted keywords and behaviour for a stored procedure language. Each RDBMS system has implemented this standard differently, so it is important to understand that moving between them will not necessarily be an easy task.

The stored procedures covered in this section are all written using the DB2 SQL Procedure Language, known as SQL PL. Stored procedures can be written in a variety of languages, including C, the Java language, and OLE, however, it is easier to illustrate how they can be used if simple SQL language examples are used instead. This is the language we'll use in this section.

Why use stored procedures?

You may want to use stored procedures in your applications for a number of reasons:

- **Reduced network traffic**: Because the stored procedures send only the procedure call and the final result across the network, not all of the interim data sets need to be sent back to the client. For applications where the client and the server are geographically distant, this can dramatically improve performance.

- **Improved performance**: The stored procedures will be executed on the server instead of on the client. In the vast majority of architectures, the server is a powerful machine that can process the SQL much quicker than the client can.

- **Reduced complexity**: By placing all of the database application code in modular units, you have isolated the application's interaction with the database to a single layer. This allows for easier testing and debugging as well as maintenance.

- **Division of responsibilities**: Having all of the data interaction isolated to the stored procedures allows the interactions to be controlled by the DBA. The DBA is the one who knows the system best, and he or she will be able to respond the quickest to any performance problems.

- **Ease of use**: Programming applications that process data using stored procedures is quite simple and allows business logic to be easily translated into application logic. By leaving the application in the data format that accesses the database, less application programming skill is needed.

When to use -- and not to use -- stored procedures

The previous panel noted a number of benefits of using stored procedures. There are, however, limitations on using stored procedures.

- Stored procedures can return any DB2 system data type. The return value of the stored procedure, however, can only be an integer.
- A stored procedure cannot be called within an SQL statement.
- Stored procedures cannot be called within triggers or functions.
- The length of a procedure is limited to 64 KB of text.

Unfortunately, the workaround for using stored procedure logic in functions and triggers is to embed the entire stored procedure logic in the function or trigger.

Stored procedures are most efficient when they are used to encapsulate a large section of related logic. You should treat your stored procedures just like application programs. If a section of code is repeated often throughout your application or procedures, then it probably makes sense to encapsulate it into a stored procedure. However, if the logic is only called once, then it makes sense to leave it as code.

You should also avoid writing very large stored procedures that perform multiple tasks, unless the stored procedure is clearly defined as doing so. You may want to put all of your database logic into a stored procedure. If you have a simple call to the database to return a result set, then creating a stored procedure for this task would be overkill. It would be better to open a cursor directly within your application.

Creating a stored procedure

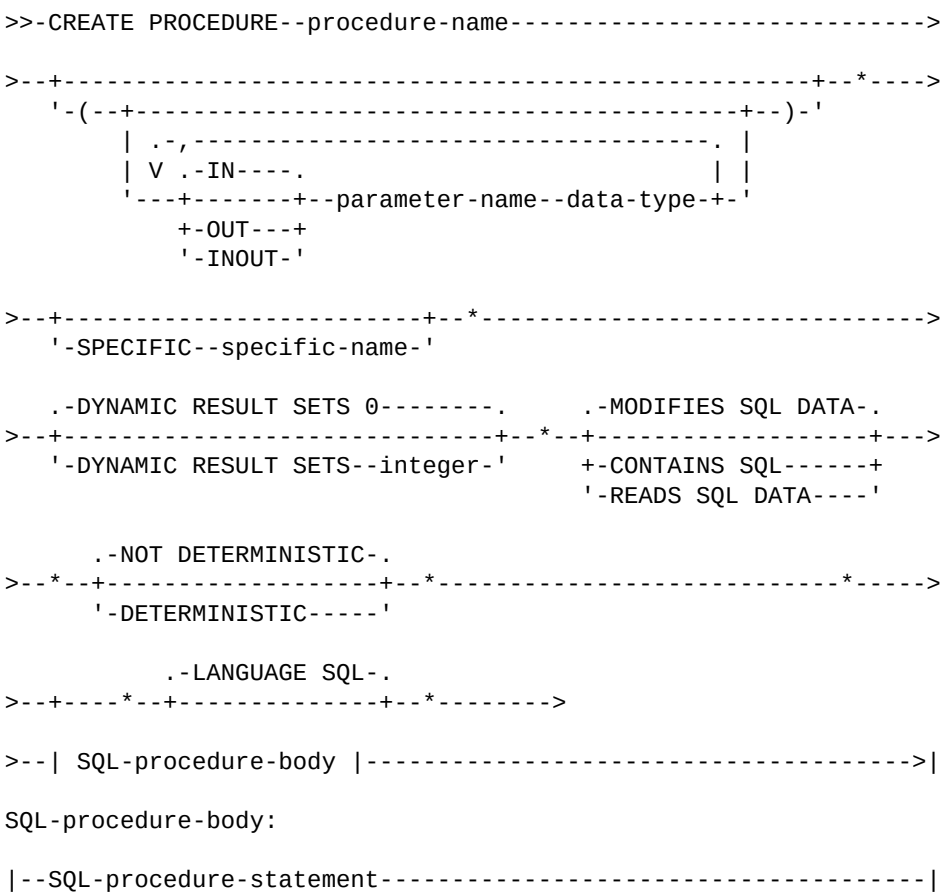
Creating a stored procedure is not a difficult process. First, you need to have a C compiler installed on your system. Stored procedures are currently translated from DB2 SQL PL into C code and then compiled as binaries. Without the compiler, you will not be able to create any new procedures, though you will be

able to take existing stored procedures from other systems and install them on your current server. (You can do this using the `GET ROUTINE` and `PUT ROUTINE` commands. More detail about the commands and the restrictions on moving procedures is provided in the DB2 online documentation.)

You can use existing free or open source compilers for compiling the stored procedures. The two most common compilers used are the GCC compiler and the trial edition of the Microsoft Visual C++ compiler (see Resources on page ?). Microsoft currently has a trial version of its compiler available that can be used for compiling stored procedures. This is for development purposes only. A full license must be purchased if you are going to use it in production.

Creating a procedure

Once you have a compiler installed, you are ready to create a procedure. The syntax is similar to that for creating a function.



The major components of the command are:

- **Parameter type**: There are three parameter types. `IN` is used for input parameters. Changes to these parameters are not passed back to the calling

application. `OUT` is used for output parameters. Changes to these parameters are passed back out to the calling application. `INOUT` is used for input and output. Changes to these parameters and their input values affect the stored procedure and the calling application.

- **Specific**: A specific name can be assigned to label a stored procedure internally. DB2 assigns a system-generated value if a name is not specified.

- **Dynamic Result Sets**: Specifies the number of results sets that will be left open and passed back to the calling program or application.

- **Deterministic**: Specifies if the stored procedure will return the same result set if the same input is given.

- **Language**: Currently, only the DB2 SQL PL language is supported for SQL procedures.

SQL statements in routines

There are four settings for stored procedures that specify how they interact with the database and what SQL they use.

- `NO SQL`: Indicates that no SQL is used in the procedure.
- `CONTAINS SQL`: Indicates that SQL statements that neither read nor modify SQL data can be executed by the stored procedure.
- `READS SQL`: Indicates that some SQL statements that do not modify SQL data can be included in the stored procedure.
- `MODIFIES SQL`: Indicates that the stored procedure can execute any SQL statement except statements that are not supported in stored procedures.

If a statement invokes a routine, the effective SQL data access indication for the statement will be the greater of:

- The SQL data access indication of the statement from the following table.
- The SQL data access indication of the routine specified when the routine was created.

The table below is taken from the DB2 documentation:

SQL Statement	NO SQL	CONTAINS SQL	READS SQL DATA	MODIFIES SQL DATA
COMPOUND SQL	N	Y	Y	Y
CONNECT	N	N	N	N

CREATE	N	N	N	Y
DECLARE CURSOR	Y	Y	Y	Y
DECLARE GLOBAL TEMPORARY TABLE	N	N	N	Y
DELETE	N	N	N	Y
DESCRIBE	Y	Y	Y	Y
DISCONNECT	N	N	N	N
DROP ...	N	N	N	Y
END DECLARE SECTION	Y	Y	Y	Y
EXECUTE	N	Y	Y	Y
EXECUTE IMMEDIATE	N	Y	Y	Y
EXPLAIN	N	N	N	Y
FETCH	N	N	Y	Y
FREE LOCATOR	N	Y	Y	Y
FLUSH EVENT MONITOR	N	N	N	Y
GRANT ...	N	N	N	Y
INCLUDE	Y	Y	Y	Y
INSERT	N	N	N	Y
LOCK TABLE	N	Y	Y	Y
OPEN	N	N	Y	Y
PREPARE	N	Y	Y	Y
REFRESH TABLE	N	N	N	Y
RELEASE CONNECTION	N	N	N	N
RELEASE SAVEPOINT	N	N	N	Y
RENAME TABLE	N	N	N	Y
REVOKE ...	N	N	N	Y
ROLLBACK	N	N	N	N
ROLLBACK TO SAVEPOINT	N	N	N	Y
SAVEPOINT	N	N	N	Y
SELECT INTO	N	N	Y	Y
SET CONNECTION	N	N	N	N
SET INTEGRITY	N	N	N	Y
SET special register	N	Y	Y	Y
UPDATE	N	N	N	Y

VALUES INTO	N	N	Y	Y
WHENEVER	Y	Y	Y	Y

Creating stored procedures

This tutorial does not cover all of the semantics of creating a stored procedure. What's most important to understand is how a stored procedure can be created from the command line. Normally a script can be executed using the following syntax:

```
db2 -tvf fileName
```

This syntax will not necessarily work for your stored procedure, depending on how line endings are handled. The different options for the `db2` command that are used with running scripts are:

- `t`: Indicates that `;` is to be used to indicate the end of a line. This allows SQL statements to be spread over more than one line in the script. If a character token is put after the `-t` option, it will be the token used for line delimitation.
- `v`: Turns on verbose mode. All of the script commands will be repeated in the script output. This is useful if you are trying to identify a failed command. When a script file becomes large, debugging can be a quite time consuming and difficult process without verbose mode on.
- `f`: Specifies the name of input file.
- `d`: Specifies that the token for the end of command is separate from the token for end of line. This is essential for stored procedures, as you need to be able to differentiate between the end of a line and the end of the procedure.

Let's look at an example procedure:

```
CREATE PROCEDURE drewProc1 (IN p_varOne INT,
                OUT p_varTwo INT)
LANGUAGE SQL
BEGIN
        DECLARE v_varUno INT DEFAULT 0;         -- (1)

        SET v_varUno = 1 + p_varOne;
        SET p_varTwo - v_varUno * p_varOne;
END
@                                                                    -- (2)
```

If the script is compiled using the command `db2 -tvf fileName`, DB2 will stop at (1) and treat it as one command, resulting in an error. To avoid this problem, you have to compile it with the command `db2 -td@ -f fileName`.

DB2 will then treat the ; character at (1) as an end-of-line character and continue processing the procedure. Only when the @ character is reached at (2) will DB2 stop processing it as one entire procedure.

Stored procedure examples

Below are a couple of simple examples of stored procedures. The first example could have been written as a function as well, since it only returns the count of the number of employees of a certain education level.

```
CREATE PROCEDURE empCount (IN p_edLevel,
                OUT p_edCount)
LANGUAGE SQL
BEGIN
        DECLARE c_cnt CURSOR FOR
                        SELECT count(*)
                        FROM employee
                        WHERE edlevel = p_edLevel;

                OPEN c_cnt;

                FETCH FROM c_cnt INTO p_edCount;
END
```

The second example is more complex and expands on the first. It checks how many employees are in a work department and gives them a larger budget if they have less than four people.

```
CREATE PROCEDURE increaseSize (IN p_size DECIMAL)
LANGUAGE SQL
BEGIN
        -- Declare returncode
        DECLARE SQLSTATE CHAR(5);

        -- Declare variables
        DECLARE v_deptCount INT DEFAULT 0;
        DECLARE v_dept CHAR(3) DEFAULT '';

        -- Declare the cursor
        DECLARE c_cnt CURSOR FOR
                        SELECT workdept,count(*)
                        FROM employee
                        group by workdept;

        -- Open the cursor
        OPEN c_cnt;

        -- Fetch from the cursor until it is empty
        FETCH FROM c_cnt INTO v_dept, v_deptCount;
        WHILE ( SQLSTATE = '00000' ) DO
                -- Check if the department has four or less people
                IF v_deptCount < 5 THEN
                        -- update the salary if they do
```

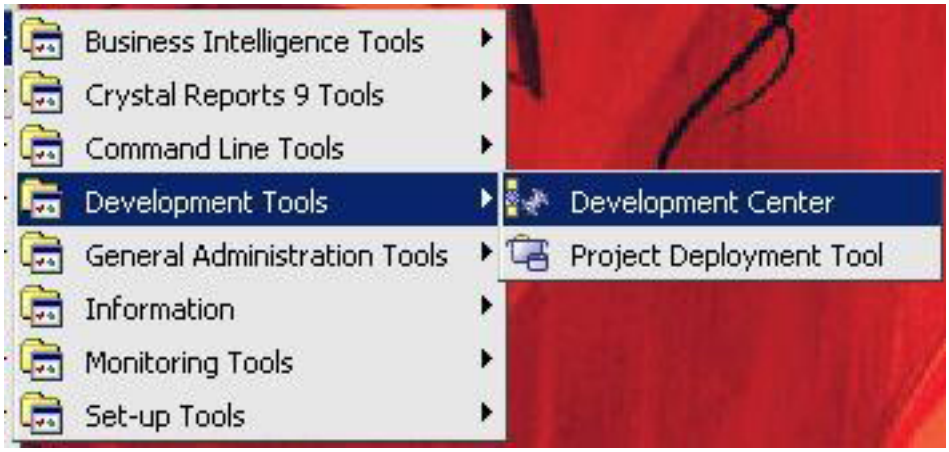
```
                        UPDATE employee
                                SET salary = salary * (1 + p_size)
                                WHERE workdept = v_dept;
                END IF;

                FETCH FROM c_cnt INTO v_dept, v_deptCount;
        END WHILE;
END
@
```

Section 5. Using the DB2 Development Center

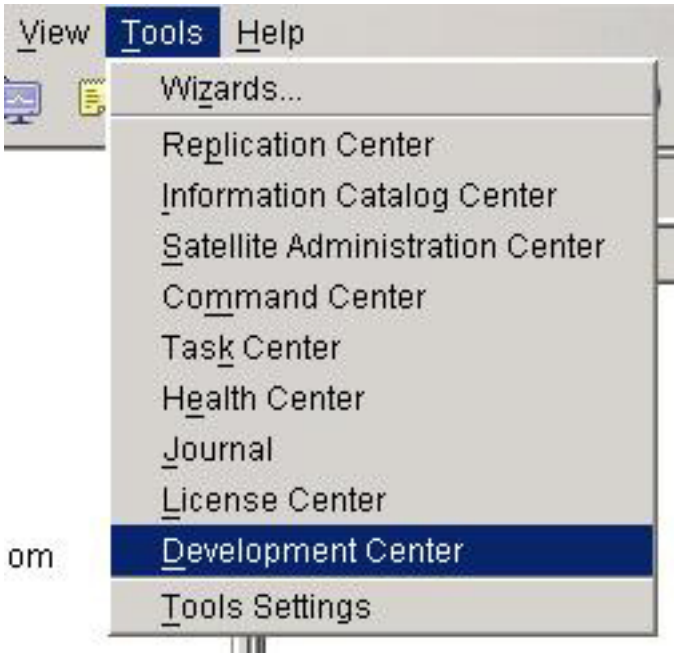
What is the DB2 Development Center?

Creating and debugging stored procedures can be a difficult task, as most common IDEs don't support SQL. To overcome this problem, IBM created the DB2 Development Center, known in DB2 UDB V7 as the DB2 Stored Procedure Builder. A great deal of functionality has been added to the Development Center (commonly referred to as the DC) since its initial release.

The Development Center is included with DB2 as a free tool. You can run it by selecting it from the IBM DB2 options in the Windows Start Menu:



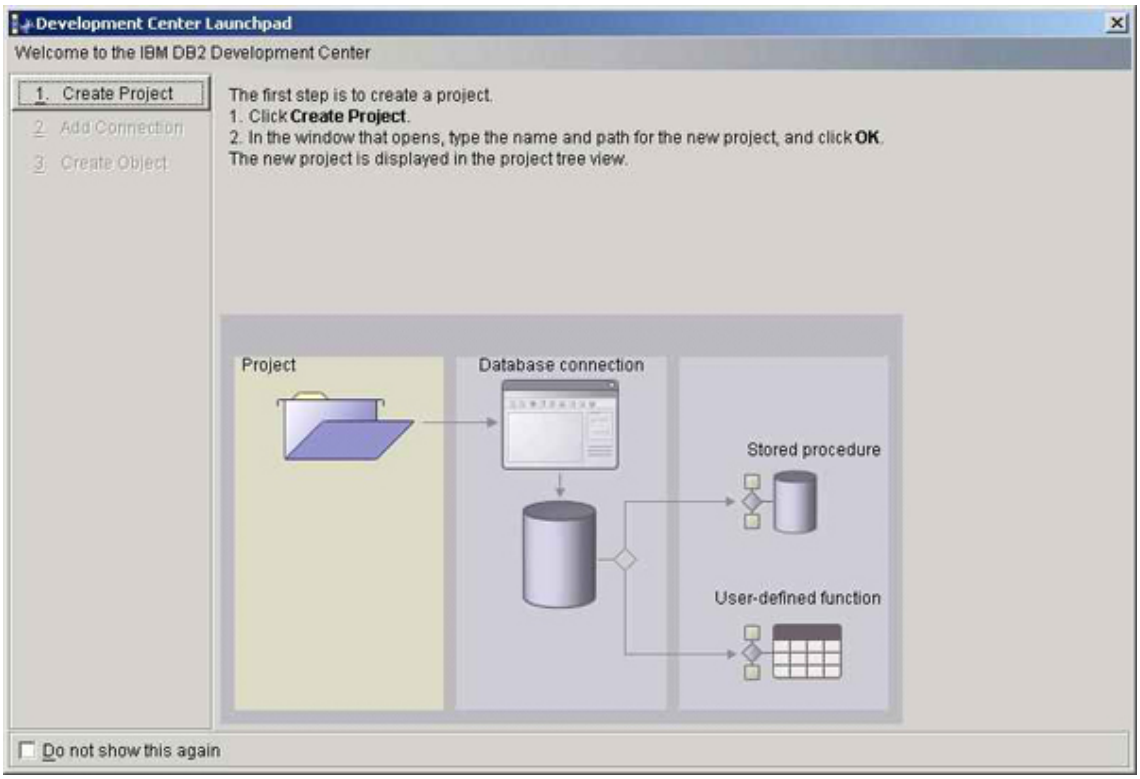
You can also run it from within any of the other DB2 GUI tools, such as the DB2 Control Center.



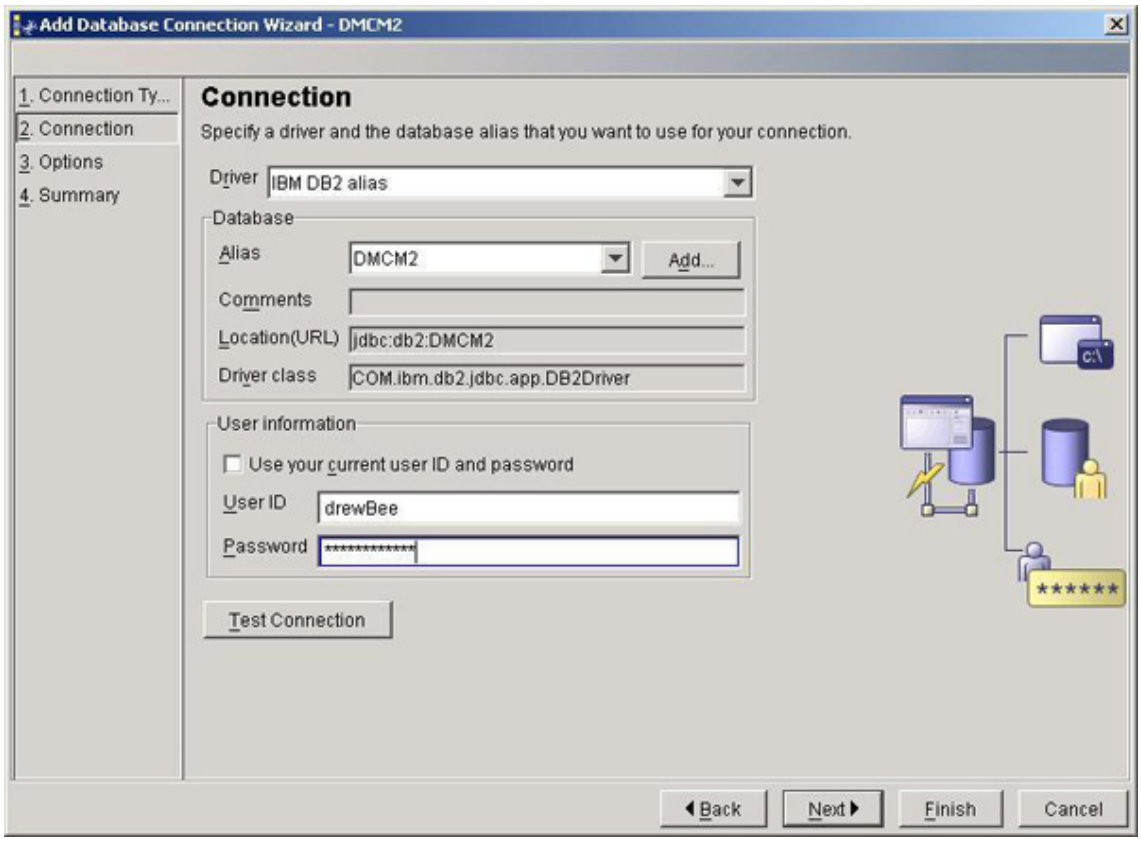
If you are going to be using the DC from a UNIX or Linux client, then you must have the X Window System or equivalent software installed to ensure that the GUI will be displayed.

Setting up a project

When you first open the Development Center, you will be asked to create a *project.* The project stores all the information about the databases you have connections to, the procedures you are working on, and the status of your project. Previously, the Stored Procedure Builder would display all of the procedures that you had in the database. By being able to pick which ones you want to work with, projects can be handled much more clearly.

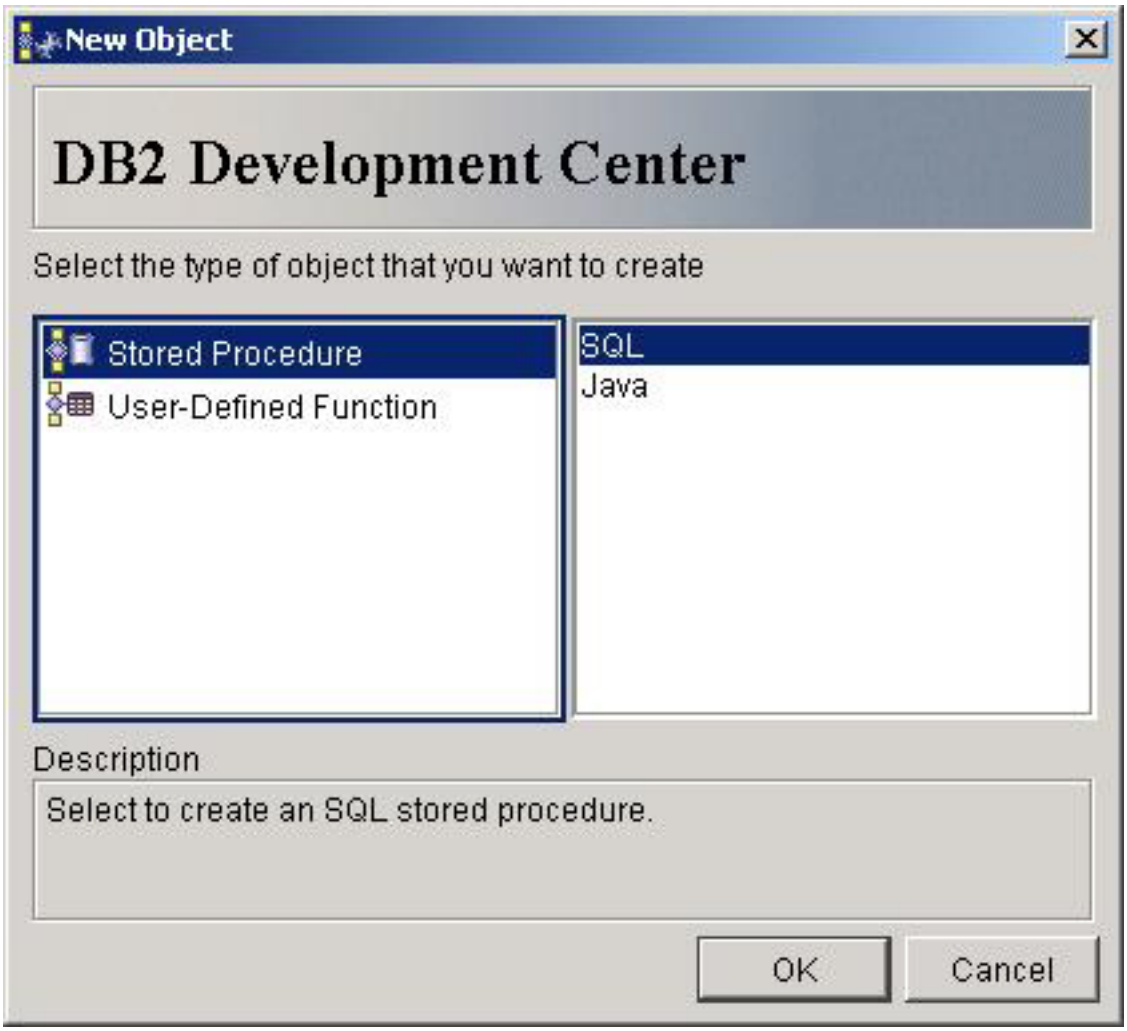


The project needs an initial connection to a database. The list of databases provided to you is taken from the databases that you have catalogued on your local client. If the required database is not in the list, you can add it by selecting the **Add** option. This catalogues the new database on the client.

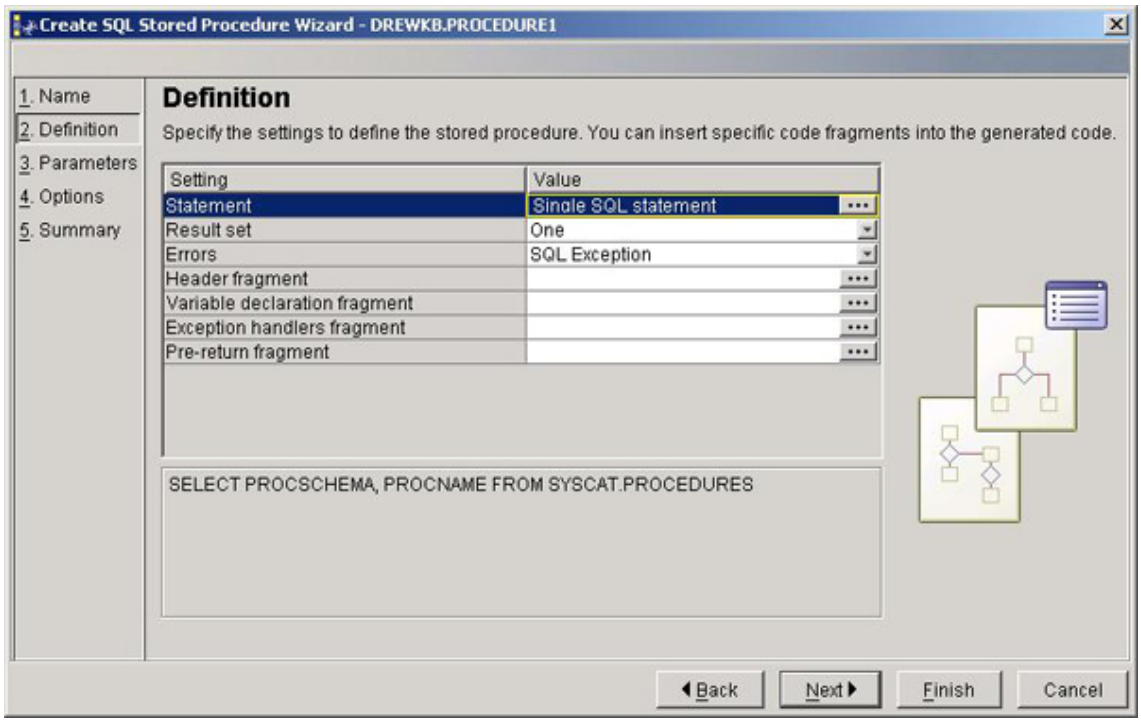


Creating an object

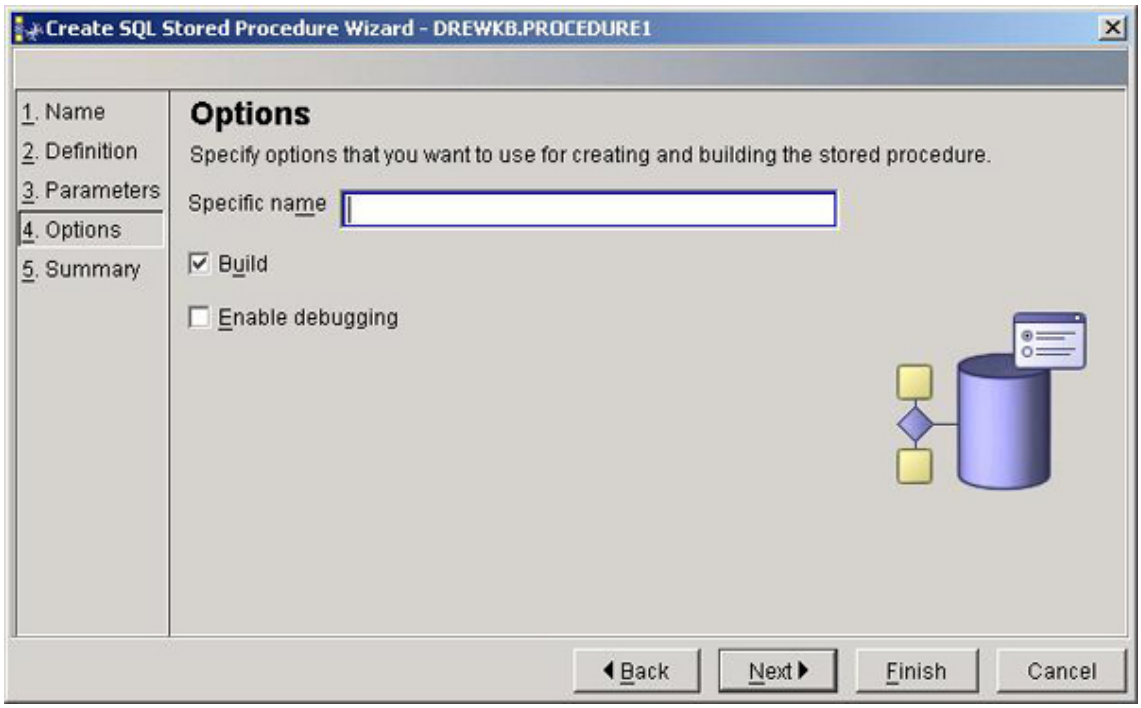
An SQL PL procedure isn't the only kind that you can build in the Development Center. You can also work with Java procedures, though only in a project working with user-defined functions. We'll cover the use of Java procedures in Advanced use of the Development Center on page44 .



If you want to build a new procedure, the DC walks you through all of the options step by step. The Object Building wizard is simple enough that even novices to SQL procedures can build a procedure, but also complex enough that experts can build fully functional stored procedures. The figure below shows the different options for the header of the stored procedure.



You should also give the procedure a specific name in case any overloading is used. (The use of specific names with stored procedures is discussed in Creating a procedure on page 27.)



Using the Development Center

There are four main views in the Development Center:

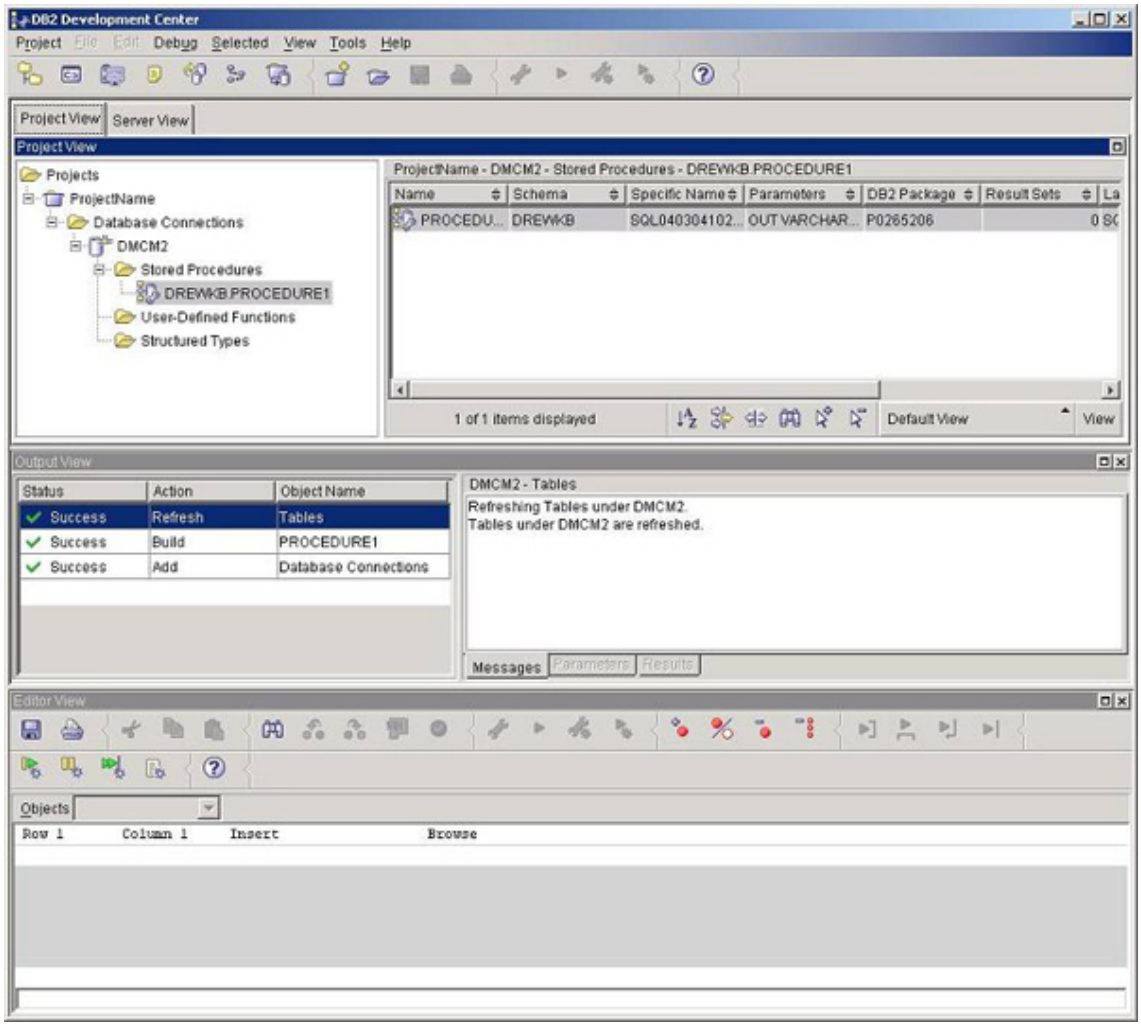
- **Project view**: Lists all of projects that are currently open, the databases they have connections to, and the stored procedures that are part of a particular project.

- **Server view**: Summarizes the items based on database connections and includes information on stored procedures, triggers, functions, tables, and views.

- **Output view**: Lists the status of the actions you have just completed or are working on.

- **Editor view**: Displays objects for editing.

The following figure shows you an overall picture of the Development Center interface.

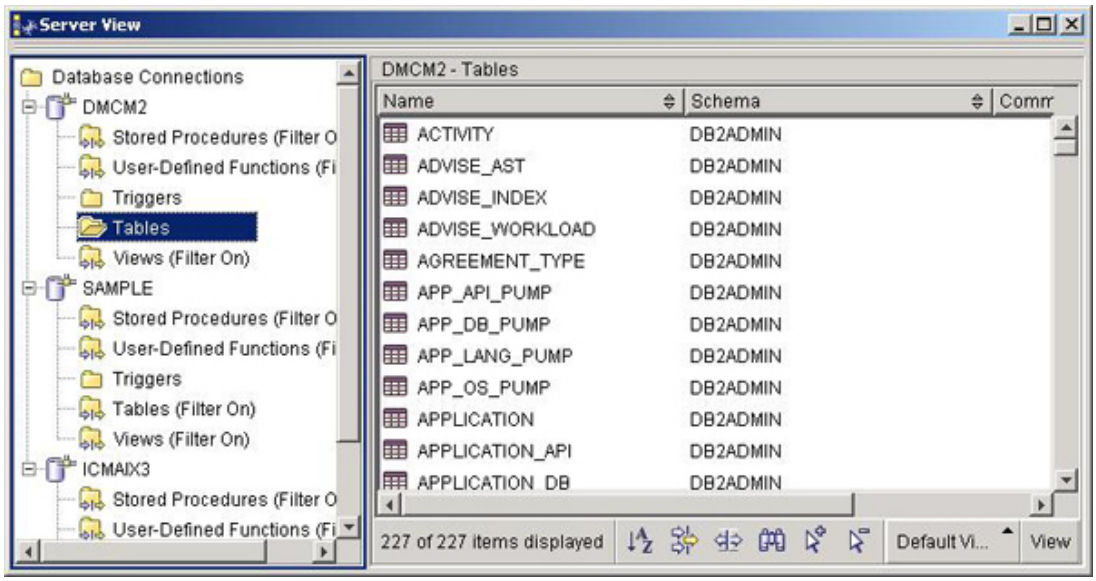


Changing the DC interface

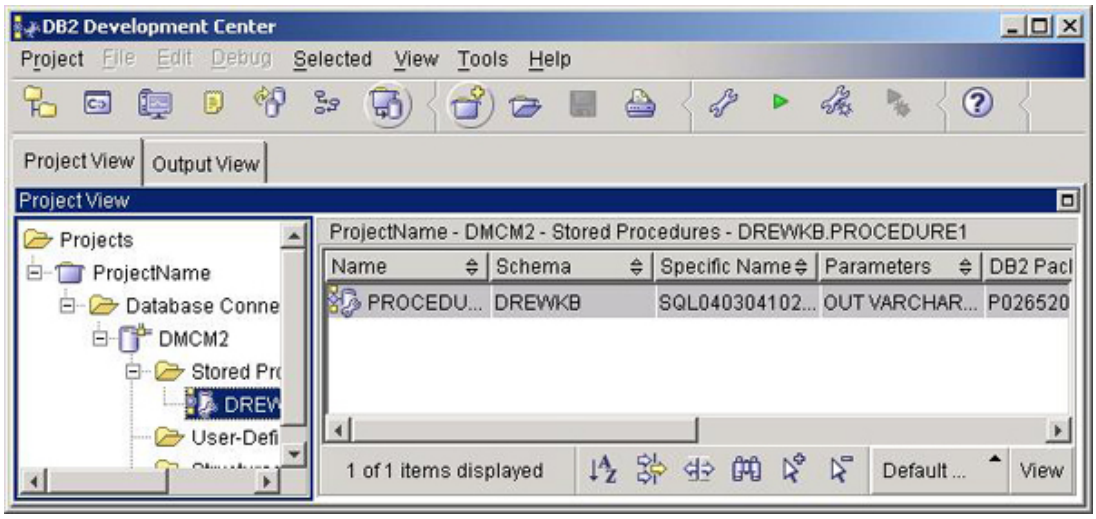
The DC is designed to be very flexible and adaptable. None of the locations of the menus or their sizes is fixed. If you hold your mouse over the dark blue menu line for any of the views, a hand-shaped icon appears. You can use this icon to drag the menu option around the screen.

The DC can display a view in a number of ways:

° **Outside of the DC**: The view can be dragged outside the DC to become a stand-alone object. This is useful for viewing a list of objects in a separate frame, which reduces the clutter of the Development Center.



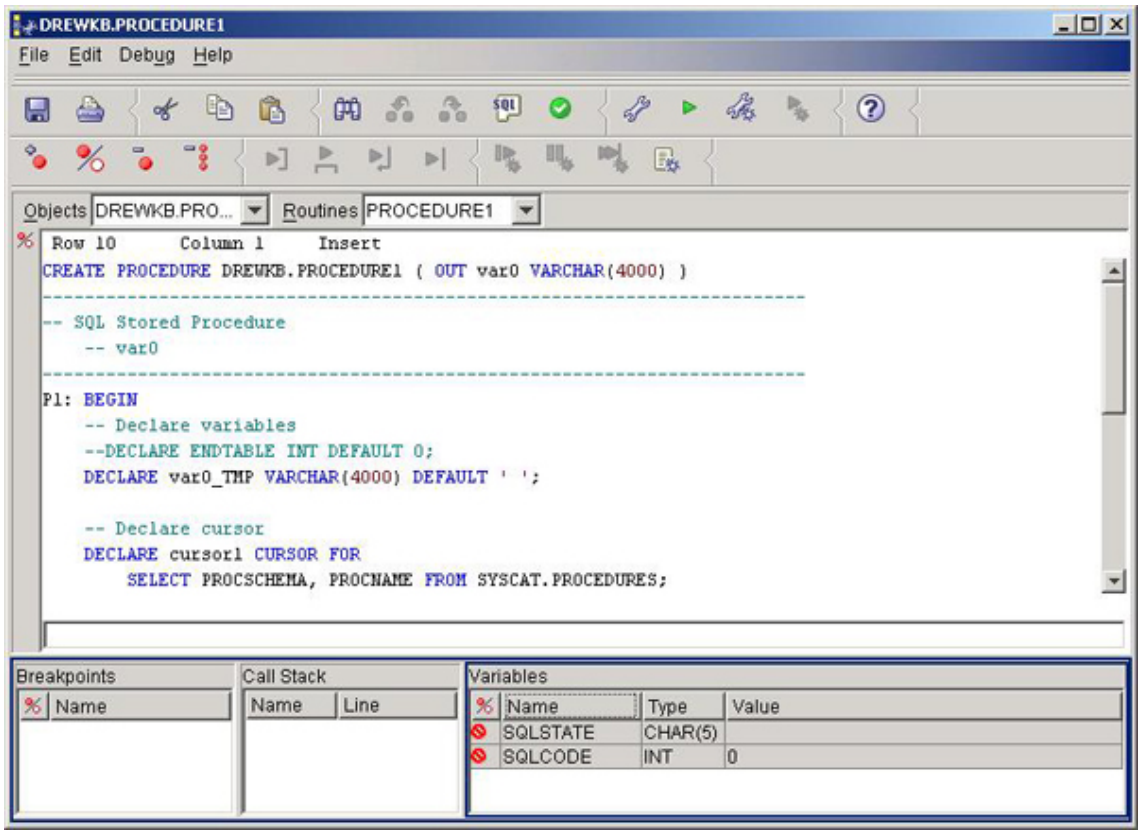
° **Tab**: The views can also be lined up as tabs. All of the information can be quickly viewed but still remain hidden when you don't need to see it. Selecting this option can be a little tricky. You have to select the view with the hand icon and move it over to the edge of another view. Move the object around the edge until you see an object appear that resembles a set of file tabs. You can then let go and the views will be combined into a set of tabs. Two or more views can be combined this way.



The views can also be reset to the default settings by selecting **View=>Reset Views** from the menu.

Working with stored procedures

The stored procedure editing window contains all the information that you need to edit, build, and debug your procedures. The main editor window is a graphical IDE that highlights the code based on the syntax. This makes writing the procedures easier.



The individual icons used for building and debugging the procedures are covered in the following panels.

Building a procedure

There are two main modes that the stored procedures can be built in: *normal build* and *build for debug.* The build for debug option allows you to debug the procedure. It automatically inserts all the required breakpoints and step-through information into the procedure. It is therefore recommended that for a production environment you ensure that your stored procedures have been built in normal mode. The debug mode significantly slows the performance of the procedures.

- **Build (1):** This command builds the stored procedure in regular mode.
- **Run (2):** This command executes the stored procedure.
- **Build in Debug (3):** This command builds the stored procedure in debug mode.

Debugging a stored procedure

One of the most important functions of the DC is the ability to step through your procedures in debug mode. This allows you to find your errors and keep track of how variables are changing. Once you have built your stored procedure in debug mode, you can add breakpoints to the code.

- **Add Breakpoint (1)**: This adds a breakpoint to a line of code.
- **Toggle Breakpoint (2)**: This changes how the breakpoints are handled, changing it from enabled to disable or vice versa.
- **Remove breakpoint (3)**: This removes the breakpoint.
- **Remove all breakpoints (4)**: This removes all breakpoints.
- **Run in Debug (5)**: This runs the stored procedure in debug mode.

Not all of the commands support breakpoints. If you try to run a program in debug mode and do not have any breakpoints, or have breakpoints on incorrect commands, the program will run normally and you will not be able to step through it.

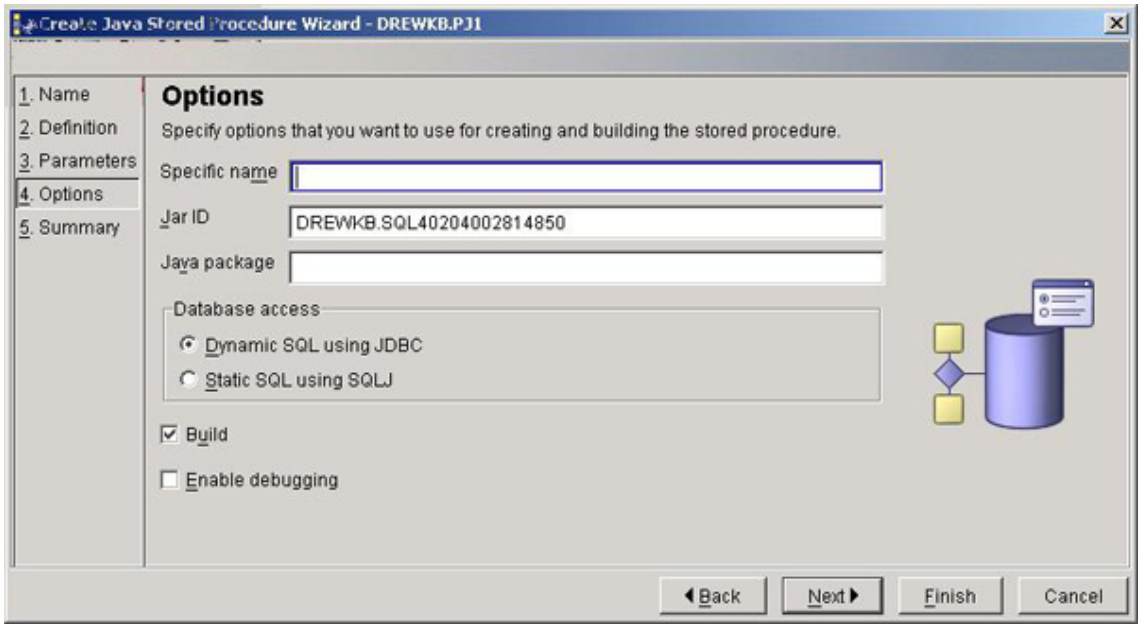
Category of statements	Statement
Statements that accept breakpoints	ALLOCATE CURSOR, ASSOCIATE LOCATORS, CASE (EXPRESSION), COMMIT, CREATE PROCEDURE, CREATE {table,view, index}, DEFAULT (VALUE), DROP {table,view, index}, ELSEIF (EXPRESSION), EXECUTE, EXECUTE IMMEDIATE, FETCH {..}, INTO, FOR v1 AS {SQLstmt}, GET DIAGNOSTICS, GOTO(LABEL), IF (EXPRESSION), RETURN(value), SELECT {..} INTO, SET (EXPRESSION), UNTIL (EXPRESSION), WHEN (VALUE), WHILE (EXPRESSION)
Statements that change variables	CALL FETCH {..}, INTO, GET DIAGNOSTICS, SELECT {..} INTO, SET
Statements that do nothing	BEGIN, BEGIN NOT ATOMIC, BEGIN ATOMIC, CLOSE CURSOR, DECLARE cursor, WITH, RETURN, FOR {sql statement}, DECLARE , var without default, DECLARE CONDITION (CONDITION) FOR SQLSTATE (VALUE) "..." , DECLARE CONTINUE HANDLER, DECLARE CURSOR, DECLARE EXIT HANDLER, DECLARE RESULT_SET_LOCATOR [VARYING], DECLARE SQLSTATE, DECLARE SQLCODE (unless there is a default), DECLARE UNDO HANDLER (unless they are entered), DO, ELSE END,

	END, CASE END, IF END, FOR END, REPEAT END, WHILE, ITERATE, LEAVE, LOOP, OPEN CURSOR, REPEAT (as a keyword alone), RESIGNAL, SIGNAL, THEN labels, i.e. P1:

Section 6. Advanced use of the Development Center

Using the Java language with stored procedures

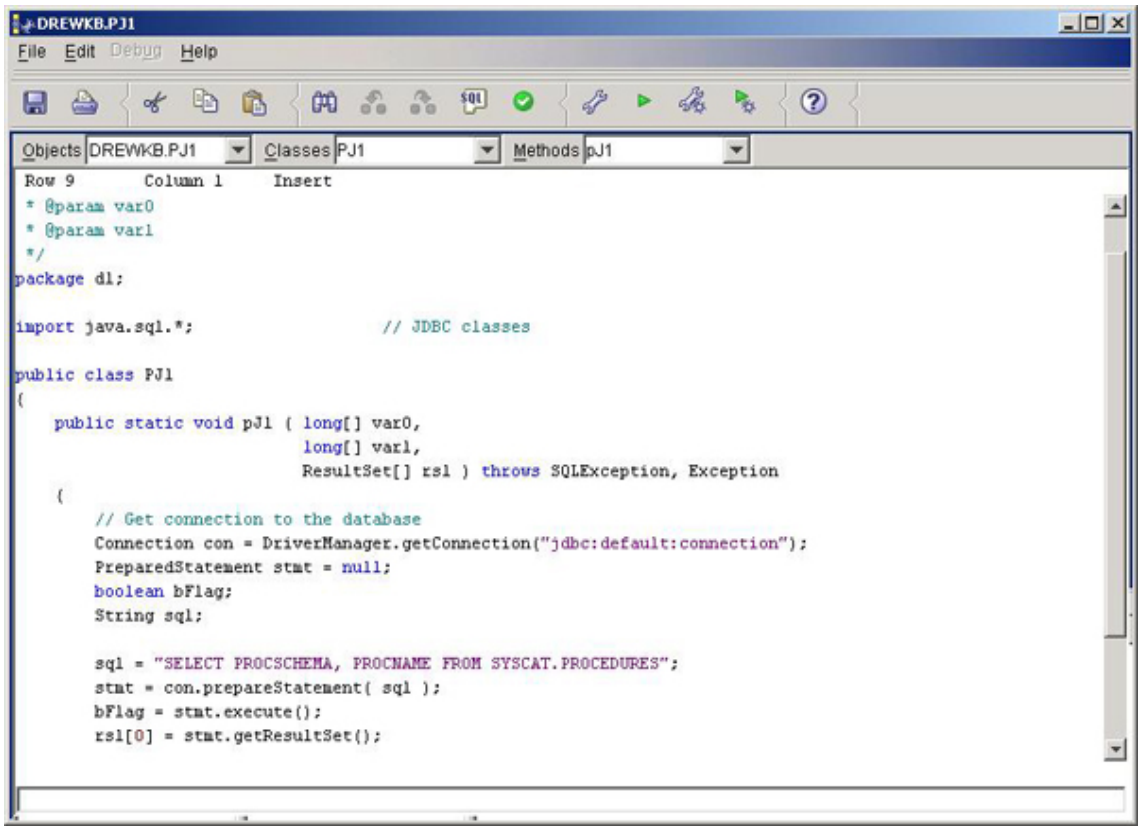
The development center allows you to work with stored procedures written in the Java language. There is even a wizard that follows very similar steps to those that you used to create an SQL procedure. The main differences become apparent when you reach the options page. At this point in the wizard, you can select what the JAR ID for the stored procedure will be and also the package name.



Stored procedures can be developed using both SQLJ (static JDBC) or standard Java code using JDBC.

Editing Java procedures

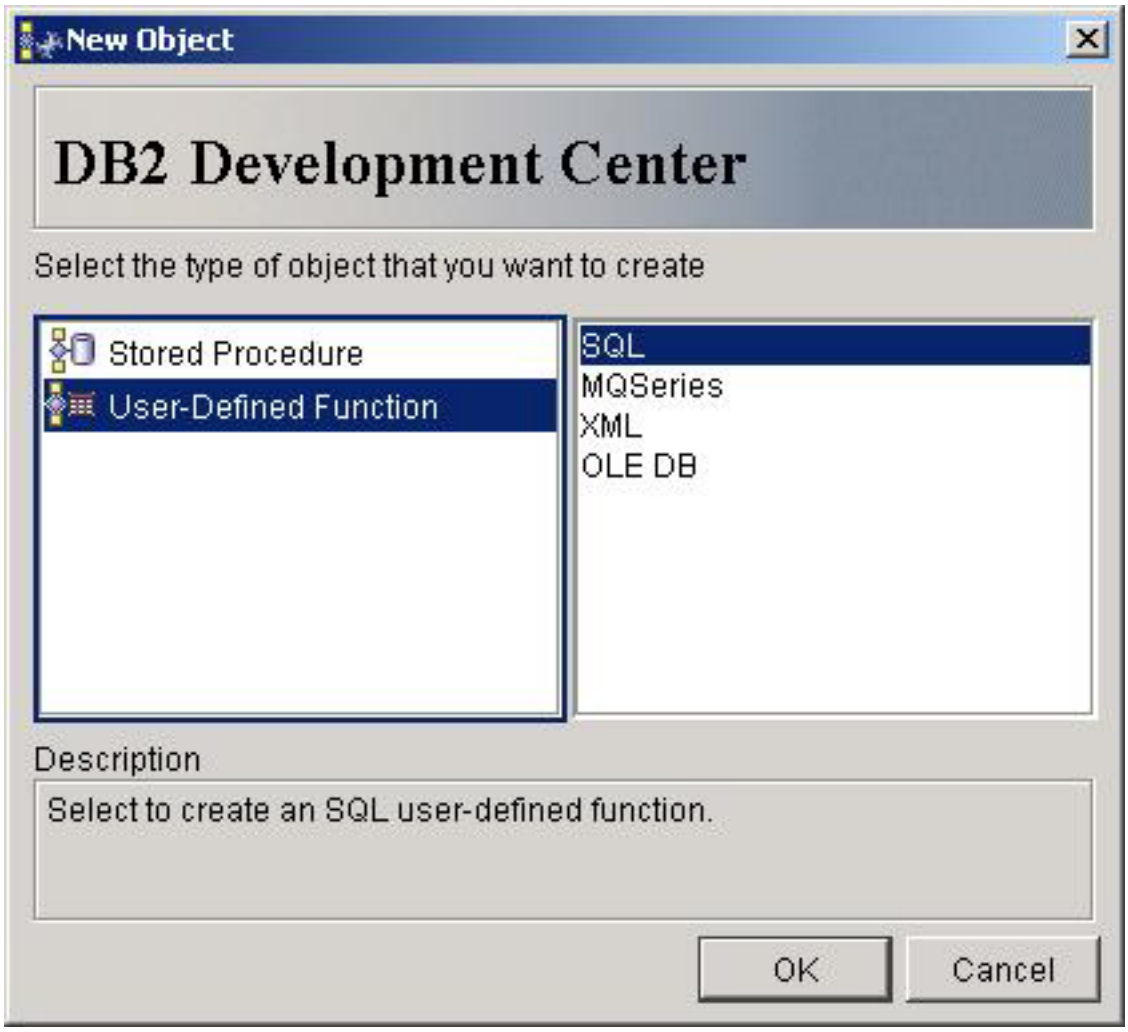
The editor window for Java stored procedures is slightly different from that for SQL PL. It does, however, offer a full editor that allows you to add and work with Java programs utilizing the JDBC interface for working with the database.



A Java procedure can also be run normally as if it were an SQL PL procedure. The dual run and debug capabilities of the DC allow you to use it for all your stored procedure testing needs.

Creating user-defined functions

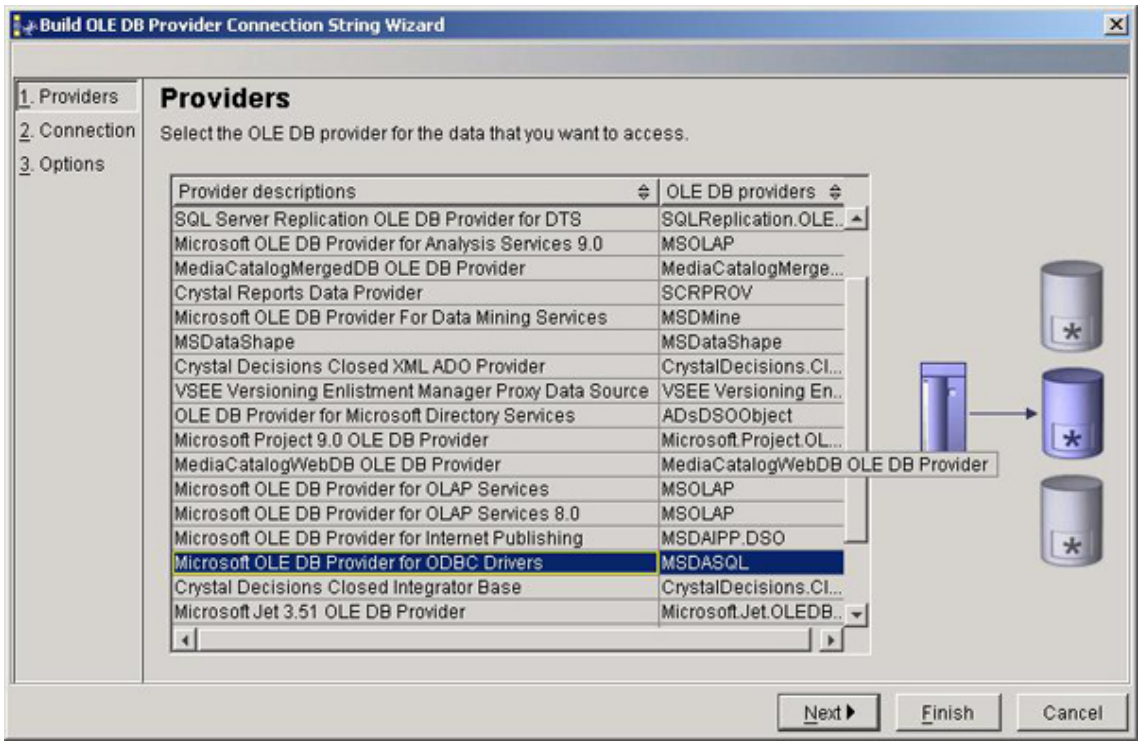
The Development Center is not limited to just stored procedures; it can also be used for user-defined functions. When you select the wizard to create an object, you can choose to build a user-defined function rather than a stored procedure. From there, you can choose a variety of programming languages that you can work with.



The list of languages that the Development Center supports is limited to a subset of the total available languages for UDFs. If you want to develop UDFs in other languages, you have to obtain the appropriate IDE and follow the steps laid out in Using user-defined functions on page 5.

Creating an OLE function

If you would like to create an OLE DB-based function, the wizard walks you through the required steps. Initially, you have to select the OLE provider you'll be using.

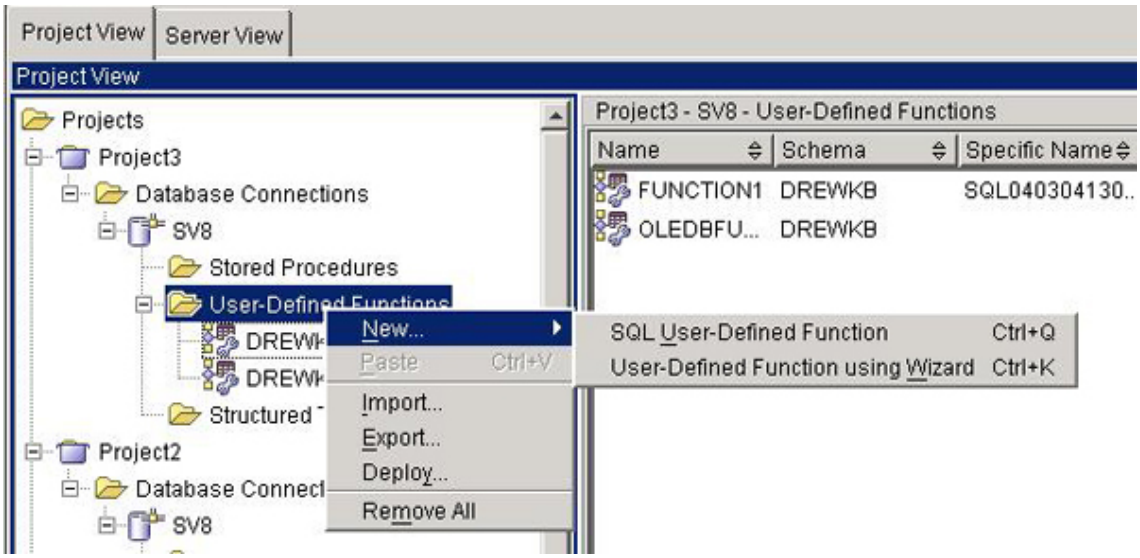


Once the provider is selected, you'll need to provide your user ID and password for the connection. This information is used to build the connection string for the OLE function.

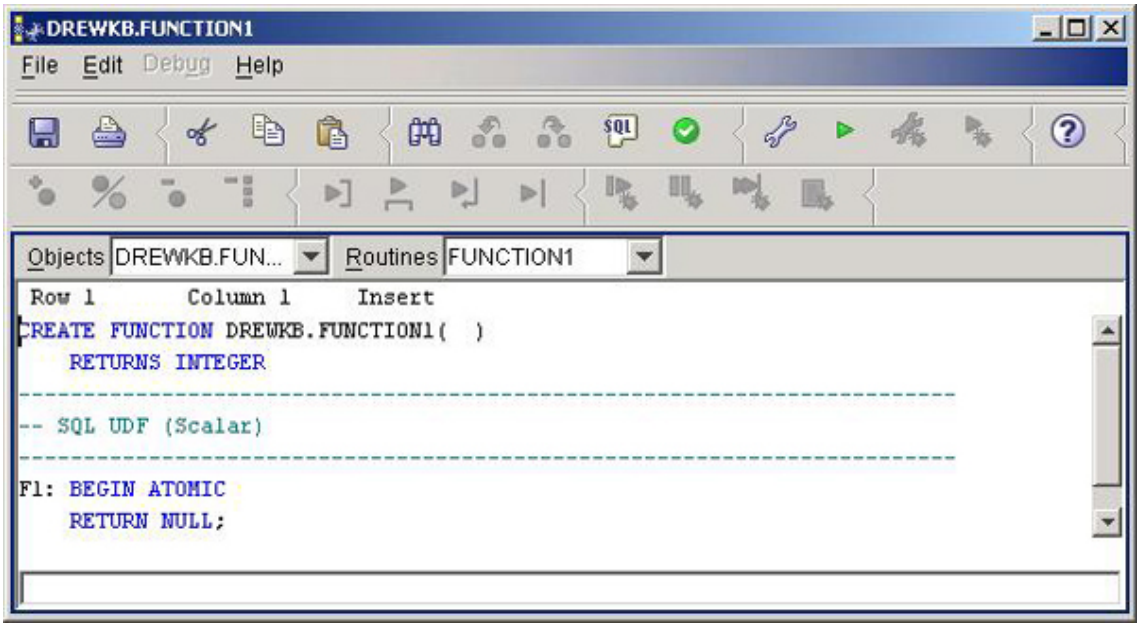
Creating an SQL function

SQL functions can be created quickly by selecting the User Defined Function folder in the Project view. You can then create the function by using the wizard,

or by having the DC quickly create a base template for you to fill in.



The UDF editing window is similar to the procedure editor, but you no longer have the debugging option.



Section 7. Conclusion

Summary

This tutorial covered how functions and stored procedures can be used in your application. User-defined functions can be used in a variety of environments and with multiple languages. Both SQL and external programs can be used to build complex functions that can be integrated into your SQL statements.

Stored procedures are becoming an increasingly important component of any application that interacts with a database. By understanding why and when stored procedures should be used, you can improve the performance and manageability of your code. Like functions, stored procedures can be written in multiple languages, including the Java language, C, and SQL PL. This tutorial focused on using SQL PL, but many companies use Java procedures to ensure portability between database platforms.

The discussion of stored procedures and functions was all tied together in the overview of the DB2 Development Center. The Development Center is a leap forward from the original stored procedure IDE, the Stored Procedure Builder. The DC allows you to build and debug stored procedures written in both SQL PL and the Java language. It now has the capacity to build user-defined functions written in SQL PL and a variety of external languages as well. You can use projects to contain all of your database objects, procedures, and functions into one modular unit. This new packaging in the DC will help you build your applications with DB2.

Resources

This tutorial will help you study for the application development exam, but there is only so much space and time to explain the many concepts. The best source of information for DB2 is the new Information Center. This Information Center was added in DB2 V8.1 Fixpak 4 and has an entirely new interface written using the Eclipse programming framework. It is quick and easy to use, and the search utility built into it is far better than any of the previous incarnations. It also has links to other sources of DB2 information on the Web, such as Google and the DB2 Technotes.

All of my references in the sections refer to this source and not to the DB2 documentation included with the product itself. The new Information Center is written by the DB2 Information Development team and is the new official reference for DB2 help. Go out and try it! I know you will like it.

The DB2 Information Center is also available on the Web at:
http://publib.boulder.ibm.com/infocenter/db2help/index.jsp

There is an excellent and in-depth reference on stored procedures available. I

am slightly biased, since I am one of the authors but I still recommend it to anyone who has to work extensively with procedures: *DB2 SQL Procedural Language for Linux, Unix and Windows*, Yip, Bradstock, Curtis, Gao, Janmohamed, Liu, and McArthur (Prentice Hall, 2002).

In Creating a stored procedure on page 26, we discussed the fact that you need a C compiler on your system in order to compile stored procedures. There are several options available to you:

- *GCC compiler*
- Microsoft is offering a free .Net compiler (and other .Net development tools) on its Web site. There are two products you need in order to use these compilers. (They also *must* be installed in the order presented here.) The first is the *.Net Framework Version 1.1 Redistributable Package*; the second is the *.Net Framework SDK Version 1.1*.

- For more information on the DB2 UDB V8.1 Family Application Development Certification (Exam 703), see *IBM DB2 Information Management -- Training and certification* (http://www.ibm.com/software/data/education/) for information on classes, certifications available and additional resources.
- As mentioned earlier, this tutorial is just one tutorial in a series of seven to help you prepare for the DB2 UDB V8.1 Family Application Development Certification exam (Exam 703). The complete list of all tutorials in this series is provided below:
  1. *Database objects and Programming Methods*
  2. *Data Manipulation*
  3. *Embedded SQL Programming*
  4. *ODBC/CLI Programming*
  5. *Java Programming*
  6. *Advanced Programming*
  7. User-Defined Routines
- Before you take the certification exam (DB2 UDB V8.1 Application Development exam, Exam 703) for which this tutorial was created to help you prepare, you should have already taken and passed the DB2 V8.1 Family Fundamentals certification exam (Exam 700). Use the *DB2 V8.1 Family Fundamentals certification prep tutorial series* to prepare for that exam. A set of six tutorials covers the following topics:
  - DB2 planning
  - DB2 security
  - Accessing DB2 UDB data
  - Working with DB2 UDB data
  - Working with DB2 UDB objects
  - Data concurrency
- Use the *DB2 V8.1 Database Administration certification prep tutorial series* to prepare for the DB2 UDB V8.1 for Linux, UNIX and Windows Database Administration certification exam (Exam 701). A set of six tutorials covers the following topics:
  - Server management

- Data placement
- Database access
- Monitoring DB2 activity
- DB2 utilities
- Backup and recovery

Check out *developerWorks Subscription* for one-stop access to a comprehensive portfolio of the latest IBM software from DB2, Lotus, Rational, Tivoli, and WebSphere, allowing you to maximize ROI and lower your labor costs, leading to superior productivity.

Feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .